In the specification:

Please amend the paragraph beginning at page 5, line 11 as follows:

A1 Fig. 10 is a block diagram of a first computer system that loosely couples a plurality of tightly-coupled processors.;

Please amend the paragraph beginning at page 5, line 13 as follows:

A2 Fig. 11 is a block diagram of a second computer system that loosely couples a plurality of tightly-coupled processors.;

Please amend the paragraph beginning at page 5, line 22 as follows:

A3 The microkernel layer 104 interfaces with the hardware layer 102. The microkernel layer 104 runs at a kernel level where the microkernel layer can execute privilege operations to allow the kernel to have full control over the hardware and user level programs. The application ~~layer~~ portion 108 and the OS command/library portion 110 run at a user level. The user level interacts with the kernel level through various systems call interfaces. The user level executes at an unprivileged execution state of the hardware and thus are executed in a restricted environment, controlled by the microkernel layer. Hence, the microkernel layer prevents simultaneously executed programs from interfering with one another either intentionally or maliciously. The microkernel layer 104 executes a non-preemptive microkernel that can run other kernels or microkernels as processes. As such, the microkernel layer 104 can protect the nature of kernels that have "piggybacked" onto the microkernel layer 104. For instance, the user can run the Unix operating system as a process managed by the microkernel layer 104.

Please amend the paragraph beginning at page 6, line 9 as follows:

A4 A process in this abstraction is very "light weight" compared to a regular Unix process. A process in this abstraction requires as little as a few hundred bytes of memory for the process descriptor and a stack. Many processes can run on each processor. Process switching time is very small because only the stack pointer and caller-saved registers need to be saved and restored. All local processes run in the same address space, so there are no memory maps to

Cont A4

switch. The microkernel layer contributes negligible CPU overhead for single-process applications as well as most multiprocessing applications.

Please amend the paragraph beginning at page 6, line 16 as follows:

A5

The microkernel layer 104 is a task-based operating system and executes a plurality of tasks, as shown in Fig. 2. Tasks 120, 122, 124 and 126 are executed in a predetermined priority sequence that is shown in Fig. 2 as priority levels 1 through priority level N. Any of the tasks can be used to execute an operating system kernel or microkernel. Thus, in the exemplary embodiment of Fig. 2, tasks 122 and 124 are processes associated with different operating systems.

Please amend the paragraph beginning at page 6, line 21 as follows:

A6

The microkernel layer 104 in effect becomes a "dominant" operating system that loads "sub-dominant" operating systems for execution. Each sub-dominant operating system environment is set up as a process that depends from the dominant operating system. The dominant operating system can run one or more functional processing operations such as networking, file system, and storage. The sub-dominant operating systems in turn run application specific processes such as SNMP decoding, Java bytecode execution or CGI scripting operations.

Please amend the paragraph beginning at page 6, line 27 as follows:

A7

The relationship between the microkernel layer 104 and the operating system tasks it supports is show in more detail in Fig. 3. In this exemplary configuration, the microkernel layer 104 executes task 1 120, the operating system microkernel task 122 and task N 126. The operating system microkernel task 122 in turn runs an operating system such as Unix and a plurality of processes executed by the operating system of task 122. For instance, the task 122 can execute a parent process 1 130 that can fork and generate one or more child processes 132 that become executed child processes 134. Upon completion of the executed child processes 134, the executed child processes become zombie processes 136 and control is returned to the parent process 130~~138~~. Similarly, the task 122 can execute a parent process 2 140 that can fork

Cont A7

and generate one or more child processes 142 that become executed child processes 144. Upon completion of the executed child processes 144, the executed child processes become zombie processes 146 and control is returned to the parent process 140~~148~~.

Please amend the paragraph beginning at page 8, line 6 as follows:

A8

A process can create a new process on the same processor by calling

pid = k_ create(func, stack_size, priority, arg);

where func is a pointer to the function that the process will execute, and arg is an argument, of type long, that is passed to func~~,~~ stack_ size specifies the size in bytes of the process' stack, and priority specifies the process' scheduling priority level.

Please amend the paragraph beginning at page 9, line 6 as follows:

A9

Fig. 5 shows components associated with a message. A message is a fixed- sized vector containing K_ MSG_ SIZE bytes of data. The first section of every message is a message type field 200 that ~~identify the~~ identifies a function requested by the message. Data is stored in a message content area 202. Messages also include private information and may need to obey special alignment required by hardware. An IPC system area 204 is used for local message administration purposes and ~~does not get~~ is not retrieved by ~~the~~ a receiving processor. Only the ~~Message Type~~ message type and the ~~Message Contents~~ data stored in the message content area of a message are retrieved by the receiving processor. In one implementation, the IPC system area 204 includes a sender PID 210, a forwarder PID 212, and a destination PID 126. The IPC system area 204 also includes a message address 218 and a message type (e.g., UNIX) 220. The IPC system area can have local values 222.

Please amend the paragraph beginning at page 9, line 13 as follows:

A10

All communication between boards is through primitives to send, receive, and reply with their options. When a process sends a message, it goes to sleep. When a reply is sent for that message, the kernel locates the message (via the address contained in the reply message descriptor) and within that message it finds the processing structure. It then links the reply

cont A10

message to the processing structure and places the process on the run queue. Once the process is running, it finds the reply message on its own processing structure.

Please amend the paragraph beginning at page 9, line 19 as follows:

Fig. 6 illustrate in more detail the use of a message in sending data to a process. Fig. 6 depicts an example message communication between a process running on CPU-1 (Process Send) and a process running on CPU-2 (Process Receive). A user-level process space 230, a microkernel layer 232, and a FIFO 234 are associated with CPU-1. A user-level process space 240, a microkernel layer 242, and a FIFO 244 are associated with CPU-2. As shown therein, the steps ~~are~~ of sending data to a process include:

A11

1. The sending process ~~(Process–S) does~~ (Process Send) implements k_ alloc_ msg() to obtain a message.

2. A message is obtained from a free message list. The microkernel layer 232 allocates space for additional messages if there are no messages in the ~~freelist~~ free message list. If a message vector cannot be allocated due to insufficient memory, k_ alloc_ msg() will panic the board. All processes running on a processor share the same address space.

3. ~~Process–S~~ The sending process creates the message.

4. Messages are sent to a PID. Therefore, the sending process obtains the PID of the receiving process ~~(Process–R)~~ (Process Receive). In general, this occurs only at boot- up, since processes never terminate.

5. ~~Process–S~~ The sending process sends the message to the PID of ~~Process–R~~ the receiving process; ~~Process–S~~ the sending process blocks waiting for the reply.

6. The ~~M16~~ microkernel layer 232 on ~~CPU–S~~ CPU-1 pokes a message descriptor (the address of the message) into ~~the~~ FIFO 244 of ~~CPU–R~~ CPU-2.

7. Poking of the message descriptor into the FIFO 244 of ~~CPU–R~~ CPU-2 causes an interrupt on ~~CPU–R~~ CPU-2.

8. ~~CPU–R~~ The microkernel layer 242 on CPU-2 fetches message descriptor from its FIFO 244.

9. ~~CPU–R does~~ The microkernel layer 242 on CPU-2 implements a k_ alloc_ msg() to allocate local space for the message.

10. ~~The receiving processor~~ The microkernel layer 242 on CPU-2 DMAs the message from the VME (Versa Module Europe bus standard) space address included in the message descriptor. The message includes the PID of the sender so that the receiving process knows to where to reply.

Cont All

11. ~~CPU-R~~ The microkernel layer 242 on CPU-2 passes the local- space address of message to ~~Process-R~~ the receiving process in the user-level process space 240.

12. ~~Process-R~~ The receiving process has been sleeping at k_ receive() (~~I.e.~~ i.e., blocked) waiting for a message; receipt of message causes ~~Process-R~~ the receiving process to become ready.

13. ~~Process-R~~ The receiving process does the processing requested in the message.

14. ~~Process-R~~ The receiving process sends a reply (this does not cause ~~Process-R~~ the receiving process to block). In this example, ~~Process-R~~ the receiving process returns to sleeping at k_ receive(), blocked until the next message arrives.

Please amend the paragraph beginning at page 10, line 17 as follows:

Fig. 7 shows a corresponding message reply process from a process. As shown in Fig. 7, the steps are:

14. ~~Process-R~~ The receiving process sends a reply (this does not cause ~~Process-R~~ the receiving process to block). In this example, ~~Process-R~~ the receiving process returns to sleeping at k_ receive(), and remains blocked until the next message arrives.

A12

15. ~~M16-R~~ The microkernel layer 242 on CPU-2 DMAs message back to the original ~~M16-S/ Process-S~~ message space of CPU-1 and does a k_ free_ msg() to free local message space.

16. ~~M16-R~~ The microkernel layer 242 on CPU-2 pokes ACK message descriptor into ~~FIFO-S~~ FIFO 234 of CPU-1.

17. Poking of message into FIFO 234 causes an interrupt on ~~CPU-S~~ CPU-1.

18. ~~CPU-S~~ CPU-1 fetches message descriptor from its FIFO 234.

19. ~~M16-S~~ The microkernel layer 232 on CPU-1 notifies ~~Process-S~~ the sending process of the reply.

20. ~~Process-S~~ The sending process has been blocked waiting for the reply; receipt of the reply causes ~~Process-S~~ the sending process to become ready.

Cont A12

21. ~~Process S~~ The sending process runs, using information in the reply message.

22. ~~Free message.~~ The message can be freed.

Please amend the paragraph beginning at page 11, line 1 as follows:

Figure 8 shows a flowchart for executing processes using the microkernel. First, a workload such as a system trap or error is initiated over a network (step 300). The process of Fig. 8 determines whether ~~any~~ one or more functional multi-processing (FMP) services are needed (step 302). ~~These~~ FMP services include services that handle NFS, CIFS, FTP or HTTP, among others. If one or more FMP services are required, the process of Fig. 8 schedules the FMP processes as required (step 304). From step 304, the process determines whether an applications services ~~are~~ is needed (step 306). If not, the resulting workload is sent to another functional processor for handling (step 308).

A13

Please amend the paragraph beginning at page 11, line 20 as follows:

Figure 10 shows a computer system 400~~100~~ that loosely couples a plurality of tightly coupled processors in collectively providing a high performance server. The system 400~~100~~ has a plurality of processors 402-408~~102-118~~, 412-418, 422-428~~122-128~~ and 432-438~~132-138~~. Each of the processors 402-408~~102-118~~, 412-418, 422-428~~122-128~~ and 432-438~~132-138~~ communicates over a high speed interconnect bus 430~~130~~. A memory array 420~~120~~ is also connected to the bus 430~~130~~. Additionally, a host processor 431~~132~~ communicates with processors 402-408~~102-118~~, 412-418, 422-428~~122-128~~ and 432-438~~132-138~~ over the bus 430~~130~~. The memory can be local to a set of multiprocessor nodes 402-408~~102-118~~, 412-418, 422-428~~122-128~~ and 432-438~~132-138~~.

A14

Please amend the paragraph beginning at page 11, line 27 as follows:

Figure 11 shows a computer system 401~~101~~ that loosely couples a plurality of tightly coupled processors, each with its own memory. As in Figure 10, the system 401~~101~~ has a plurality of processors 442-448~~102-118~~, 452-458~~122-128~~ and 462-468~~132-138~~ that communicates over the high speed interconnect bus 430~~130~~. A memory subsystem 470~~103~~ is locally connected to multiprocessor nodes 442-448~~102-118~~, while memory subsystems 480~~105~~

A15

Cont A15

and 490~~107~~ are locally connected to multiprocessor nodes 452-458~~122-128~~ and 462-468~~132-138~~, respectively.

Please amend the paragraph beginning at page 12, line 3 as follows:

A16

In Figures 10 and 11, the interconnect bus 430~~130~~ may be a GTL+ bus, or may be a computer bus such as a PCI bus, a SCSI bus, or a Scalable Coherent Interface (SCI) bus that is a distributed interconnect bus on both GTL and SCI. The interconnect between nodes can be a local area network or a wide area network (LAN/WAN).

Please amend the paragraph beginning at page 12, line 7 as follows:

A17

In one embodiment, the bus 430~~130~~ is a 100Mhz Slot 2 system bus that enables processors such as the Pentium II Xeon processors to be "multiprocessor ready." The bus 430~~130~~ has a synchronous, latched bus protocol that allows a full clock cycle for signal transmission and a full clock cycle for signal interpretation and generation. This protocol simplifies interconnect timing requirements and supports 100Mhz system designs using conventional interconnect technology. Low-voltage-swing AGTL+ I/O buffers support high frequency signal communications between many loads. In this embodiment, the processor supports ECC on the data signals for all L2 cache bus and system bus transactions, automatically correcting single-bit errors and alerting the system to any double-bit errors such that mission-critical data ~~are~~ is protected. The processor also supports full Functional Redundancy Checking (FRC) to increase the integrity of critical applications. Full FRC compares the outputs of multiple processors and checks for discrepancies. In an FRC pair, one processor acts as a master, the other as a checker. The checker signals the system if it detects any differences between the processors' outputs.

Please amend the paragraph beginning at page 12, line 20 as follows:

A18

In a second embodiment using the SCI bus, the interconnect bus 430~~130~~ may be deployed using a number of topologies, including a ring configuration where subsystems are connected as a ring that is not hot-pluggable. Alternatively, the interconnect bus 430~~130~~ may be a multi-ported switch where each subsystem is on its own SCI ring and therefore can be

Cont A18

hot-plugged. Additional port switches can be used to allow the system to improve the bandwidth. The standard SCI interconnect uses five meter point -to-point cabling with two fifty-pin very high density Small Computer System Interface (SCSI) style connectors for both the input and output of the SCI interconnect bus 430~~130~~.

Please amend the paragraph beginning at page 12, line 28 as follows:

A19

Also attached to the interconnect bus 430~~130~~ can be a host processor 431~~132~~. The host processor 431~~132~~ runs an operating system such as Windows-NT, available from Microsoft Corp. of Redmond, Washington, or Solaris UNIX operating system, available from Sun Microsystems of Mountain View, California. The host processor 431~~132~~ provides a platform for network and system administration, backup and archive operations, database management, and other applications. Functions such as network information services (NIS) and network lock manager (NLM) can also be executed on the host processor 431~~132~~.

Please amend the paragraph beginning at page 13, line 5 as follows:

A20

The interconnect bus 430~~130~~ supports booting of processors from the host processor 431~~132~~ or a master control processor. Generally, an on-board Basic Input/Output System (BIOS) initializes the processors on the bus 430~~130~~ and configures it to participate on the bus 430~~130~~. From there, the presence of all processors is detected by the host or control processor 431~~132~~, where a configuration utility takes over, as described in more detail below.

Please amend the paragraph beginning at page 13, line 10 as follows:

A21

To further improve reliability, other components in the system 400 of Figure 10, such as the processors 402-408~~102-108~~, 412-418, 422-428~~122-128~~ and 432-438~~137-138~~, may monitor the status of the host or control processor 431~~132~~ and determine when the host or control processor 431~~132~~ is inoperative. If the host processor 431~~132~~ is hung, the processors 402-408~~102-108~~, 412-418, 422-428~~122-128~~ and 432-438~~137-138~~ can force the host or control processor 431~~132~~ to reboot. In this event, the processors retain any state information the host or control processor 431~~132~~ requires, such as the state of the network interface cards while the host or control processor 431~~132~~ boots. New status messages are saved and forwarded to the host or

control processor 431~~132~~ after the reboot is complete. New mount request and NIS queries are serviced as soon as the reboot is complete. In a similar manner, in the event that one of the processors 402-408~~102-108~~, 412-418, 422-428~~122-128~~ and 432-438~~137-138~~ fails, the computer system 400~~100~~ of Figure 10 continues to operate without failing.

Please amend the paragraph beginning at page 13, line 21 as follows:

As shown in Figure 10, the computer ~~server~~ system 400~~100~~ is a loose~~ly~~ coupling of processors that cooperate with each other in performing server-related functions, for example, network processing, file processing, storage processing, and application processing. Due to the loosely coupled nature of the multiprocessor nodes, processors 402-408~~102-108~~, for example, can reboot on their own due to a failure and still come back to serve their designated functionalities. The heterogeneous coupling of the processors 402-408~~102-118~~, 412-418, 422-428~~122-128~~ and 432-438~~132-138~~ provides a user with the flexibility to grow the processing power of the computer ~~server~~ system 400~~100~~ as required for a specific application. For instance, certain applications require more processing power for network processing and thus more processors should be dedicated toward network processing. Other applications may require more processing power for file and storage processing and more processors should be dedicated toward these tasks.

Please amend the paragraph beginning at page 14, line 2 as follows:

The hybrid multi-processing or heterogeneous coupling of processors of Figures 10 and 11 allows the user to robustly grow the processing power of the computer ~~server~~ systems 400~~100~~ and 401~~101~~. Each processor in the n-way processors can be a:

1. Network processor;
2. File Processor;
3. Storage Processor;
4. Network and File Processor;
5. Network and Storage Processor (SAS);
6. Storage and File Processor;
7. Network, Storage and File Processor (NAS); or

Cont A33 8. Application Processor.

Please amend the paragraph beginning at page 14, line 19 as follows:

Each configured file processor has a metadata cache that contains file management information, including a directory name look-up table, among others. The directory name look-up table is used to speed up directory look -ups, as Unix file system (UFS) directories are A34 flat and much be searched sequentially. Further, the directory name look-up table maintains hits and misses for short file names. In the directory name look up the structures are kept in a least recently used (LRU) order and maintained as a hashed table.

Please amend the paragraph beginning at page 14, line 30 as follows:

The network processors provide all protocol processing between the network layer data format and an internal file server format for communicating client requests to other processors in the system. Only those data packets that cannot be interpreted by the network processors, for example client requests to run a client-defined program on the server, are transmitted to the host A35 or control processor 431~~132~~ for processing. Thus the network processors, file processors and storage processors contain only small parts of an overall operating system, and each is optimized for the particular type of work to that it is dedicated.

Please amend the paragraph beginning at page 15, line 10 as follows:

The application processor can run any off-the-shelf operating system. This processor can A36 also run specific applications. For example, the application processor can run dynamic loading of web pages or process voice and video mail management or can run Solaris or NT and can handle generic applications.

Please amend the paragraph beginning at page 15, line 14 as follows:

The architectures of Figures 10 and 11 are advantageous in that the host or control processor 431~~132~~ provides a single point of administration for system utilities and tools, including monitoring, updating, backing-up and tuning software. The architectures further takes A37 advantage of processors that are dedicated and optimized for specific tasks. As function-specific

caches are provided with each processor, through-put is further increased. Additionally, the dedicated processors can transfer data with relatively low communication overheads. The resulting system is powerful, scalable and reliable enough to allow users to consolidate their data onto one high performance system that can provide data more quickly and reliably than a conventional client/server architecture.

Please amend the paragraph beginning at page 15, line 23 as follows:

Fig. 12 shows a block diagram of a computer system 500. The computer system 500 has a plurality of processors 502-508. Each of processors 502-508 has an associated voltage regulation module (VRM) 523~~522~~, 526, 528~~532~~ and 532~~534~~, respectively. Additionally, processors 502 and 504 have Level 2 (L2) caches that are supported by an L2 VRM 524. Similarly, processors 506 and 508 have L2 caches that are supported by an L2 VRM 530. Moreover, processors 502-508 communicate over a high speed host bus 520. In an embodiment, the host bus 520 is a GTL+ bus operating at 100 MHZ. Also connected to the host bus 520 is a memory input output controller (MIOC) 550. The memory input output controller 550 controls a memory array 552 over a high speed bus that may also be a GTL+ bus.

Please amend the paragraph beginning at page 16, line 6 as follows:

Also connected to the PCI-A bus 556 is a programmable interrupt device (PID) 564. The PID 564 controls an advanced processor interrupt controller (APIC) bus 518. The bus 518 communicates with each of CPUs 502-508. Interrupts are managed and broadcasted to local APICs in each of processors 502-508. The PID 564 monitors each interrupt on each PCI device, including PCI slots 566-570 in addition to compatible interrupts IRQ0-IRQ15, and on occurrence of an interrupt, sends a message corresponding to the interrupt across a three wire serial interface to the local APICs. The APIC bus 518 minimizes interrupt latency for compatibility between the interrupt sources. The PID 564 can also supply more than 16 interrupt levels to ~~processes~~ processors 502-508. The APIC bus 518 has an APIC clock and two bidirectional data lines. The interrupts can be disabled and all processor nodes can poll each I/O device for its status. For example, each processor node can poll a status bit of a receive/transmit buffer on the NIC to determine whether a packet has been received or transmitted.

Please amend the paragraph beginning at page 16, line 18 as follows:

A30

The PCI-A bus 556 is also connected to a narrow small computer system interface (N SCSI) interface 558. The N SCSI interface 558 in turn controls one or more data storage devices 559. The narrow SCSI host adaptor may be a Symbios SYM53C810AE, that contains a high performance SCSI core capable of fast 8-byte SCSI transfers in single-ended mode.

Please amend the paragraph beginning at page 16, line 22 as follows:

A31

Also connected to the PCI-A bus 556 is a bridge to an ISA input output subsystem (PIIX4) 572. The PIIX4 provides an IDE floppy controller 574, a universal serial bus (USB) controller 576, a baseboard management controller (BMC) 578, a flash memory 582 for BIOS ROM and extension storage, and an ISA slot 584. The PIIX4 572 also communicates with a Super Input/Output device 586 that drives a floppy drive 588, a keyboard/mouse port 590, a parallel port 592 and one or more serial ports 594.

Please amend the paragraph beginning at page 16, line 28 as follows:

A32

The PXB 554 also provides a second PCI bus (PCI-B) 600. The PCI-B bus 600 has a plurality of PCI-B slots 602-608~~602, 608~~ with PCI-B slot 608 connected to NIC 609. Also, PCI-B bus 600 is connected to a wide SCSI (W SCSI) controller 610. The wide SCSI controller 610 may be a Symbios SYM53C896 dual channel LVD/SE (Ultra 2/Ultra) SCSI controller. The wide SCSI controller 610 in turn drives one or more data storage devices 612.